Research Article · *Transactions in GIS*, 2013, 17(5): 724–741

Out-of-Core GPU-based Change Detection in Massive 3D Point Clouds

Rico Richter, Jan Eric Kyprianidis and Jürgen Döllner

Hasso-Plattner-Institut, Potsdam University

Abstract

If sites, cities, and landscapes are captured at different points in time using technology such as LiDAR, large collections of 3D point clouds result. Their efficient storage, processing, analysis, and presentation constitute a challenging task because of limited computation, memory, and time resources. In this work, we present an approach to detect changes in massive 3D point clouds based on an out-of-core spatial data structure that is designed to store data acquired at different points in time and to efficiently attribute 3D points with distance information. Based on this data structure, we present and evaluate different processing schemes optimized for performing the calculation on the CPU and GPU. In addition, we present a point-based rendering technique adapted for attributed 3D point clouds, to enable effective out-of-core real-time visualization of the computation results. Our approach enables conclusions to be drawn about temporal changes in large highly accurate 3D geodata sets of a captured area at reasonable preprocessing and rendering times. We evaluate our approach with two data sets from different points in time for the urban area of a city, describe its characteristics, and report on applications.

1 Introduction

Advances in 3D scanning technology (e.g. LiDAR, PhotoSynth, see Snavely et al. 2006, 2008) enable fast capturing of physical objects and urban sites by means of 3D point clouds. 3D point clouds reflect the geometry of the scanned target by a discretized representation and can be used directly (e.g. simulation, analysis) or as a base for reconstructing 3D models. The precision, handling, availability, and cost effectiveness of 3D scanning systems are constantly improving and allow for applications in a growing number of areas. In the scope of GIS, 3D point clouds are traditionally applied for planning, monitoring (Schneider 2006, Monserrat and Crosetto 2008, Trinder and Salah 2011, Kang and Lu 2011), and construction of 3D models (Zhou and Neumann 2008). In particular, 3D point clouds are used to automatically derive base 3D city models such as 3D building models and terrain models – automatic reconstruction methods allow for more and more automatic and fast generation of complex 3D geovirtual environments (Lafarge et al. 2010). A key technical requirement for GIS represents the ability to store, manage, and process these massive sets of 3D point clouds captured by 3D scanning systems, for example in the case of scans of large cities or landscapes. Efficient storage, management, and processing of the data, however, represent complex, time-consuming tasks. Due to limited memory and CPU resources, applications frequently can only use reduced data sets, i.e. they cannot take full advantage of the available data.

Address for correspondence: Rico Richter, Hasso-Plattner-Institut, Universität Potsdam, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany. E-mail: rico.richter@hpi.uni-potsdam.de

Acknowledgements: This work was funded by the Federal Ministry of Education and Research (BMBF), Germany within the InnoProfile Transfer research group “4DnD-Vis”. We would like to thank virtualcitySYSTEMS and the “Stadtvermessungsamt Frankfurt am Main” for providing LiDAR data.

The need for efficient handling of massive 3D point clouds is intensified because many applications demand frequent scans and simultaneous use of scans taken at different points in time. For example, in the context of urban planning, the continuation workflows applied to virtual 3D city models can benefit from the difference analysis between scans. The process of identifying and marking regions or points that have most likely changed in the actual physical world (Butkiewicz et al. 2008) is known as *change detection* and is, in combination with the visualization of the entire 3D point cloud data, essential for performing quality control, evaluating the acquired data and updating 3D models and contents (Matikainen et al. 2004, Gröoger and Plüumer 2009). In the case of 3D point clouds with more than 10^9 points, straightforward approaches for change detection and real-time rendering turn out to be computationally not feasible. Existing solutions typically thin out or rasterize the 3D point cloud to reduce the data size (Van Gosliga et al. 2006). This, however, leads to a loss of accuracy and makes it difficult to match the original unfiltered data with the results of the change detection.

In this article, we present an approach to process 3D point clouds of arbitrary size and to detect changes in sets of massive 3D point clouds without having to thin out or rasterize the data. We introduce a new out-of-core spatial data structure that stores data sets acquired at different times and ensures fast access to subsets of the stored 3D point clouds. In particular, the technique supports the subdivision of the data into spatially arranged parts, which enables efficient distribution of workloads to multiple CPU cores or the GPU. Based on the out-of-core spatial data structure, we present and evaluate three different optimized parallel computation schemes. The first is a multi-core CPU-based implementation, while the other two perform the computations on the GPU by using GPGPU and CUDA technology. Our evaluation of these approaches shows that the CUDA implementation performs best due to its ability to leverage the high-performance on-chip memory, also known as shared memory, in a flexible way. The techniques also support the real-time visualization of change detection results used to explore and analyze the results interactively (Kreylos et al. 2008). Interactive access to data and results helps to draw conclusions about temporal changes in large highly accurate 3D data sets more easily. For this purpose, the results of the change detection are stored as attributes in the 3D point cloud data structure. The rendering is performed by a point-based 3D rendering system that has been extended to handle these attributes.

2 Related Work

Change detection in the scope of virtual 3D city models is typically performed with polygonal geometry derived from 3D point clouds. Approaches using point-to-mesh and mesh-to-mesh algorithms were introduced by Besl and McKay (1992). The need for an explicit surface model generally requires a time-consuming preprocessing to reconstruct polygonal models (Vosselman et al. 2004). In Butkiewicz et al. (2008) all points were projected to a triangulated surface model derived from a 3D point cloud to detect changes in the urban structure. The accuracy of this approach depends on the resolution and quality of the surface model and, in contrast to our approach, is only applicable to 2.5-dimensional geodata. Gosliga et al. (2006) presented a deformation analysis for a tunnel, acquired with a terrestrial laser scanner. However, the calculation is only performed on 1% of the 3D point cloud data due to the interpolation based on a regular grid used to perform the calculation. By contrast, our approach directly uses the raw data without thinning out or rasterizing the 3D point clouds.

Point-to-point comparison approaches have the advantage that they can directly operate on 3D point clouds. A theoretical and computational framework for global comparison of uniformly sampled 3D point clouds was presented by Mémoli and Sapiro (2004). A further framework for 3D point clouds with divergent point data distributions, for example resulting from terrestrial laser scans, was presented by Girardeau-Montanut et al. (2005). They addressed the problem of missing surface information due to occlusion during the data acquisition process and solved it with visibility maps to mark regions with missing surface information for the change detection process. Barber et al. (2008) derived 3D point cloud data from aerial imagery and airborne LiDAR systems organized in an octree data structure to identify significant changes in the urban area. Leite et al. (2009) presented a system to determine K-nearest neighbors (KNN) for 3D point clouds on the GPU. The data is organized as a grid data structure on the GPU that can be evaluated in real-time. Their system is able to process data in real-time, but is limited to data sizes that fit into GPU memory. By contrast, our system only needs to find the nearest neighbor, but has to deal with massive data sets. Therefore, our data structure is designed to perform out-of-core swapping mechanisms between GPU and CPU as well as CPU and secondary storage without the requirement to evaluate the data in real time. Heinzle et al. (2008) presented a flexible hardware processing unit for 3D point clouds with a focus on fundamental and computationally expensive operations. In our approach there is no need to do complex operations on 3D point clouds.

For the visualization of 3D point clouds, several point-based 3D rendering techniques have been developed in the past. Popular techniques include Surfels (Pfister et al. 2000) and QSplat (Levoy and Rusinkiewicz 2000). A high-quality rendering technique that makes further assumptions about the availability of additional data, such as density information and normals, was presented by Botsch et al. (2005). The computation of this data requires lengthy post-processing of scanned data sets and, therefore, is not feasible for massive 3D point clouds (Wimmer and Scheiblauer 2006). However, for the visualization of change detection results, no assumptions about additional data or data limits can be made. Our approach extends the concepts of Richter and Döllner (2010) to render massive 3D point clouds with out-of-core techniques at interactive frame rates.

3 System Overview

A schematic overview of our system and data processing pipeline is shown in Figure 1, and can be divided into two central parts:

(1) The processing starts with the calculation of the differences between a 3D point cloud called a *target cloud* and another 3D point cloud called a *reference cloud* that is used as a reference object. We use the distance between the target point and the closest point of the reference cloud as the metric for the degree of change. For each point in the target cloud the Euclidean distance to the closest point in the reference cloud is calculated and stored as an attribute of the corresponding point in the target cloud. For small 3D point clouds with only a few million points this computation is straightforward. For massive data sets, however, it becomes unfeasible. Our system is able to process data sets of arbitrary size by utilizing a specialized out-of-core data structure that allows for efficient organization and access to the data. Details about the change detection are explained in Section 4.

(2) The second central part of our system is responsible for visualizing the results using a rendering system for massive 3D point clouds. The attributed 3D point clouds are organized

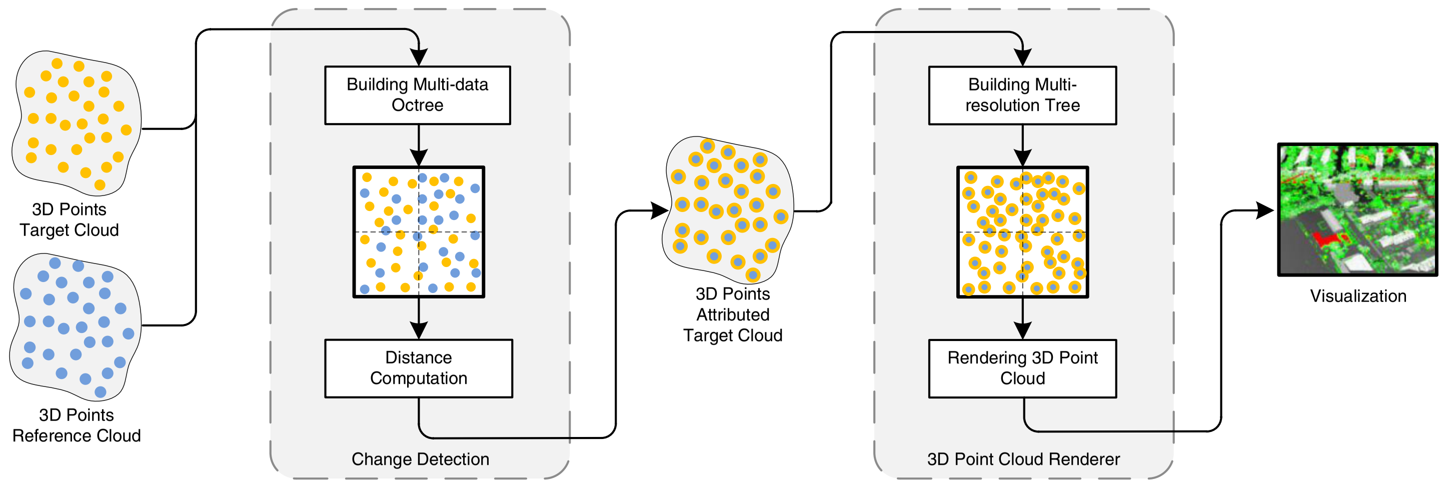


Figure 1 Illustration of the system architecture showing change detection and rendering components

in a multi-resolution tree structure that enables the interactive rendering and exploration of the data. The distance attributes represent the amount of change. They are mapped during the rendering to color attributes for each point. This allows the visualization to be customized according to the data characteristics and the application. The rendering system is described in detail in Section 5.

4 Change Detection

A straightforward approach to detect changes in 3D point clouds would be to determine the minimum distance for all 3D points of the target cloud to points of the reference cloud by simply computing the distance between all possible pairs. However, this is computationally unfeasible due to the quadratic complexity of the algorithm, which becomes particularly relevant when dealing with huge amounts of laser scan data.

4.1 Out-of-Core Multi-Data Octree

The spatial data structure in our approach is a fundamental component of the overall change detection process. It can be applied to:

- Massive 3D point clouds
- Multiple 3D point clouds
- Rapid preparation and preprocessing computations
- Fast access to sub-clouds of 3D point clouds

Our *multi-data octree* is designed to fulfill these requirements; it extends the concepts of an octree data structure and manages 3D point clouds with different characteristics. Out-of-core concepts for the multi-data octree allow the fast processing for data that exceeds the available main memory. In contrast to quadtrees, octrees can handle arbitrary 3D point clouds, e.g. from airborne, mobile, and terrestrial laser scans, which differ in their horizontal and vertical spatial distribution and density. A kD-tree is not applicable, since the subdivision of the space depends on the arrangement of the point data; it would be difficult to subdivide the space to construct a balanced kD-tree containing more than one 3D point cloud. Moreover, the preprocessing times to prepare the data structure would increase due to the need to sort the data along the longest axis for each tree level to perform a correct spatial subdivision. The decision to store multiple data sets in a single tree structure is motivated by the following reasons. The memory required to arrange the 3D point clouds is reduced, since structure information needs to be stored only for one data structure. Also the management, swapping and selection of points for multiple data sets can be performed faster because the distance calculation algorithm, in general, requests data for the same spatial part of the tree structure. The structure of the multi-data octree is illustrated in Figure 2.

Each node of the multi-data octree is either an inner node, with up to eight child nodes, or a leaf node containing 3D points of the target and reference cloud. Inner nodes subdivide the space in up to eight sub-spaces of uniform size. The root node represents a bounding box enclosing all 3D points of the target and reference cloud. The subdivision of nodes is controlled by the number of 3D points in the spatial volume of the node. If this amount is above a fixed threshold, a subdivision is performed and the node becomes an inner node. Otherwise no further subdivision is performed and the 3D points of the target and reference cloud are stored in the leaf node. Inner nodes contain information about their child nodes and the number of

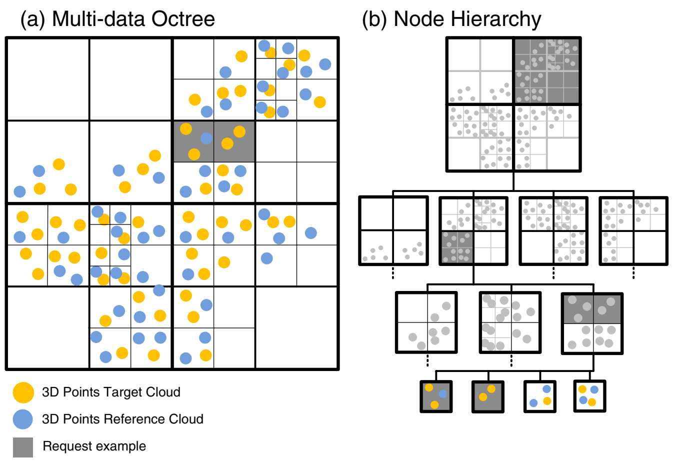


Figure 2 (a) Illustration of the multi-data octree filled with target and reference points. (b) Node hierarchy resulting from the 3D point distribution

3D points of the reference and target cloud in the represented sub-tree. This is necessary to estimate the memory requirements for each part of the multi-data octree when loading data into main memory.

The preparation of the multi-data octree is performed in a preprocessing step. If the number of 3D points of the target and reference cloud exceeds the capacity of the main memory, a direct calculation of the overall multi-data octree is not possible. In this case, the data is subdivided, using secondary storage, according to the hierarchy of the octree cells until the sub-clouds fit into main memory. For these sub-clouds the multi-data octree is prepared in main memory and serialized to secondary storage. Afterwards, all prepared parts of the multi-data octree are merged on secondary storage to build the multi-data octree containing all 3D points of the target and reference cloud.

The access to the data is performed with data requests based on a bounding volume. These volumes will be referred to as *in-memory chunks* containing the multi-data octree structure with all inner and leaf nodes that are inside the requested bounding volume or that intersect it. The time to load the in-memory chunks into main memory depends on the multi-data octree depth and the maximal capacity of leaf nodes. The total number of 3D points in an in-memory chunk also depends on the maximum capacity of leaf nodes, since all 3D points of intersected leaf nodes are selected to avoid a bounding volume test for each 3D point. This is performed to improve the loading times for in-memory chunks.

The multi-data octree used in this work is similar to the multi-resolution octree used for the visualization of 3D point clouds by Richter and Döllner (2010). In contrast to our work, the multi-data octree does not store data from multiple 3D point clouds but instead is designed to store multiple levels of detail for a single 3D point cloud. Using levels of detail for 3D point clouds is well-suited for visualization and will be further discussed in Section 5.

4.2 *Distance Computation*

To perform the distance computation, it is necessary to access the data as fast as possible to reduce processing time. Different types of memory, e.g. secondary storage, main memory, GPU device memory, and GPU registers differ in latency times to access the data and available capacity. For example, billions of 3D points can be stored on secondary storage, but access times are typically in the range of milliseconds. In contrast to that, GPU device registers can store only a view hundred 3D points, but allow access within a few clock cycles. In order to maximize the potential of available computing resources, it is therefore necessary to perform a subdivision and partitioning of the data to perform the computation in memory with low capacity but adequate access times.

The partitioning of the data is performed using the multi-data octree and is illustrated in Figure 3. The multi-data octree is divided into volumes that contain a subset of data that fits into available main memory. These volumes are referred to as in-memory chunks and their spatial size is derived from the size of the multi-data octree nodes (Figure 3a). All in-memory chunks together cover the overall volume of the multi-data octree. Special cases occur for target points close to the boundaries of in-memory chunks, because the closest reference point could be located in a adjacent in-memory chunk. To enable a correct computation, the volume of the in-memory chunk is extended to include all points from the reference cloud within a defined margin around the volume. This is illustrated in Figure 3b for an in-memory chunk transferred from the multi-data octree to main memory.

The in-memory chunks are further subdivided into small volumes called *calculation chunks*. These are used as cache to avoid frequent requests to the multi-data octree, which are required to determine the reference points for a given target point. Similar to the in-memory chunks, calculation chunks are also extended to guarantee a correct computation (Figure 3c). The size of the boundary extension depends on the application domain and the point density. Calculation chunks contain all necessary data for self-contained processing and enable, in particular, parallel processing. The size of the calculation chunks is adjusted to match the computation algorithm and device.

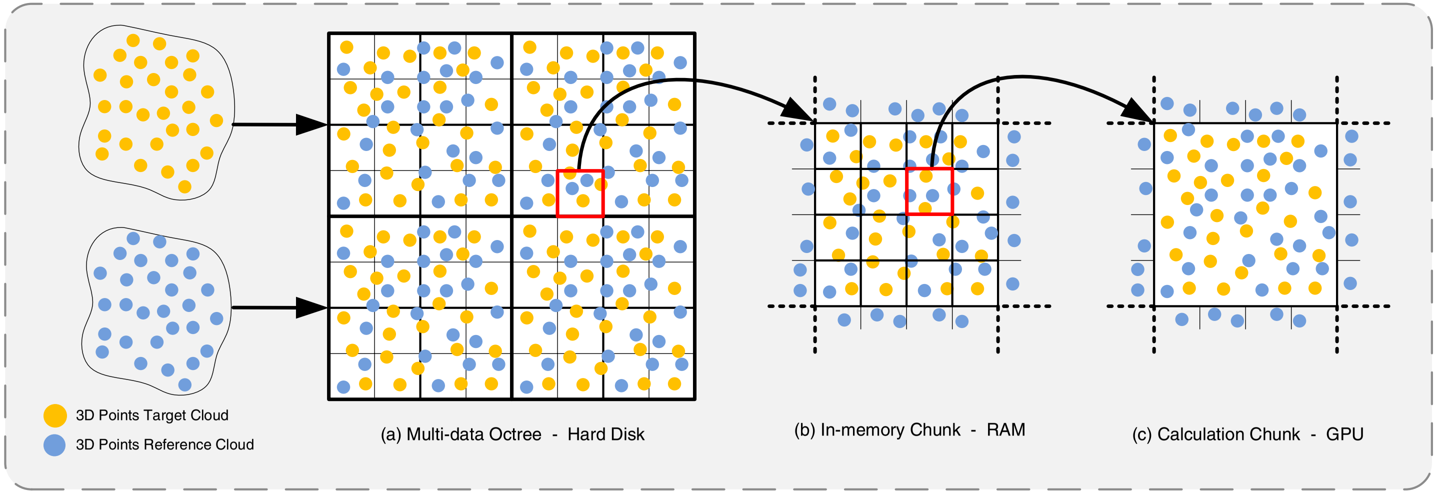
In the following we describe several implementations to perform change detection in a parallel way using multiple CPU cores and modern GPU technology. A multi-core, a GPGPU and a CUDA implementation are presented and analyzed regarding performance and hardware issues.

4.2.1 CPU Computation

The single-core CPU implementation processes all in-memory chunks in sequential order. For each in-memory chunk all calculation chunks are determined and the computation is also performed sequentially. For a given target point the minimum distance to points of the reference cloud is computed by iterating over all reference points in the calculation chunk. This computation can be performed in parallel using multiple threads to take advantage of multiple CPU cores.

4.2.2 GPGPU Computation

Graphics processing units (GPUs) were originally developed for fast processing of graphics primitives in a massively parallel way using multiple processors and dedicated memory directly on the device. Software development for GPUs is typically performed using domain specific programming languages such as the OpenGL shading language (GLSL), HLSL, or Cg. These languages have been designed for computer graphics purposes. However, a clever design of algorithms and their adaption to the available graphics primitives, such as vertex buffers and



3D Points Target Cloud
3D Points Reference Cloud
(a) Multi-data Octree - Hard Disk
(b) In-memory Chunk - RAM
(c) Calculation Chunk - GPU

Figure 3 (a) Illustration of the memory hierarchy of the multi-data octree on the hard disk; (b) In-memory chunk that is resident in main memory; and (c) Calculation chunk that is used for the computation on the GPU textures, allows GPUs to be used for other tasks as well, and has led to the development of a separate branch called general-purpose GPU (GPGPU) computing (Luebke et al. 2005).

Our GPGPU implementation of the distance calculation is based on GLSL. The selected reference points of the calculation chunk are encoded in a texture and transferred as a texture buffer object (TBO) to the GPU. The target points are transferred as a vertex buffer object (VBO), which is then rendered using a single rendering pass. The distance computation is performed in the vertex shader by computing the distances to all reference points, which are accessed by using texture fetches. Finally, for each target point the minimum distance is stored in a buffer object and the buffer object transferred back to the CPU.

The parallel computation for each target point is performed by the GPU without the possibility of controlling the threads or the used memory of the GPU directly. The frequent texture fetches turn out to be the bottleneck of this approach since each element of the VBO needs to access all elements of the TBO. In contrast to the multi-core CPU implementation, the GPGPU implementation requires only 1–2% of the computation time due to the highly parallel nature of the GPU.

4.2.3 CUDA Computation

NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing architecture that, in contrast to shading languages, provides a more flexible programming API for general purpose computations on GPUs. Interesting key features of CUDA are the ability to read and write arbitrary device memory addresses and faster transfers of data between CPU and GPU. Moreover, CUDA enables access to high-performance on-chip memory, called shared memory, that is shared by a collection of threads. Shared memory allows for the implementation of customized user-managed caching strategies, resulting in higher bandwidth and throughput (NVIDIA Corporation 2011).

Modern NVIDIA GPUs consist of several single-instruction multiple-data (SIMD) multiprocessors, each having a large number of cores. An NVIDIA GTX 480, for example, has 15 multiprocessors with 32 cores. Threads are organized into blocks and blocks are organized into a grid. A multiprocessor executes one block at a time. A warp is the set of threads executed in parallel and currently consists of 32 threads. A hardware-based thread scheduler manages scheduling threads across the available multi-processors. The threads are lightweight, enabling the scheduler to perform a zero-cost immediate context switch to another thread if a thread is waiting for memory access (Sanders and Kandrot 2010).

Similar to the GPGPU approach, our CUDA implementation first transfers all target and reference points of the calculation chunk to the device memory of the GPU. For each multiprocessor one block with 1,024 threads is scheduled. The target points are divided up between the blocks and are then assigned uniformly to the threads. For all assigned target points, the threads compute the minimum distance to the reference points, by first reading the target point and then reading all reference points from device memory. This calculation scheme is comparable to the GPGPU implementation and shows similar processing performance. However, compared to the time required for the actual computation, device memory access is rather slow. For the performance this is critical, because each thread needs to read all reference points for all assigned target points.

A better approach is to first load portions of the reference points into the low-latency shared memory of the multiprocessor. In contrast to device memory, which has high latency, shared memory allows fast access, but is limited to below 48 kB in size (Farber 2011). The reference points of the computation chunk are therefore, again, subdivided into smaller portions

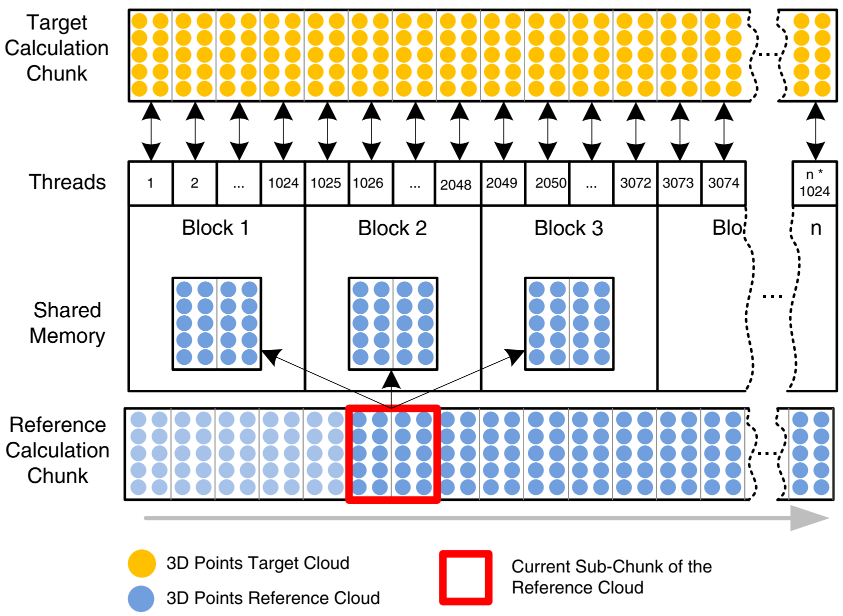


Figure 4 Illustration of the change detection computation for a calculation chunk on the GPU

that fit into shared memory. The computation now runs in two phases that are repeated until all data has been processed. First, all threads collectively load a portion of the reference points into shared memory, resulting in the device memory being read only once instead of many times. Then, after a synchronization of the threads, each thread computes for all assigned target points the distance to all reference points in shared memory and updates the minimum distance that is kept in local registers. After all reference points have been processed, the computed minimum distance is written to device memory and then transferred to the CPU. The change detection using CUDA is illustrated in Figure 4.

Processing the calculation chunks can be performed asynchronously to avoid blocking of CPU and GPU, since preparation, selection and transfer of the next calculation chunk can be performed during the computation of the current calculation chunk.

4.3 Performance Evaluation

The performance evaluation is performed for different processing levels of the overall workflow, from the processing of a single calculation chunk up to the processing of a data set with a few billion 3D points. We begin with comparisons of the distance computation for a single calculation chunk using the presented multi-core CPU, GPGPU and CUDA implementations. After that, we evaluate the processing of an in-memory chunk using the multi-data octree with different size limits to determine the calculation chunks. Finally, we evaluate the performance of the overall workflow starting with the preparation of the out-of-core multi-data octree up to the final attributed 3D point cloud using an example data set for the urban region of a city. All tests and measurements were performed on an Intel Xeon CPU with 2.66 GHz and 6 GB main memory. The GPU calculations were performed on an NVIDIA GeForce GTX 480 with 1536 MB device memory.

Table 1 Performance of processing calculation chunks with different sizes measured in seconds

#TargetPoints	#Reference Points	#Distance Calc. in Mill.	Single CPU	Multi-core CPU	GPGPU	CUDA
20,000	20,000	400	7.25	1.75	0.20	0.05
40,000	40,000	1,600	29.40	6.86	0.32	0.08
60,000	60,000	3,600	65.57	15.53	0.39	0.13
80,000	80,000	6,400	116.50	27.78	0.54	0.20
100,000	100,000	10,000	182.45	43.55	0.74	0.29
150,000	150,000	22,500	410.77	98.76	1.32	0.59
200,000	200,000	40,000	728.36	178.33	2.35	1.03

Table 2 Performance of the CUDA implementation for different sized calculation chunks in seconds

Limit Target Points	#Calc. Patches	Mean Target Pts.	Mean Ref. Pts.	Calc time in ms	Select. time in ms	Total time in sec
10,000	1,821	2,745	29,132	21,352	2,148	23.50
20,000	1,056	4,734	36,120	16,461	1,589	18.05
30,000	547	9,140	43,595	12,636	1,392	14.03
40,000	327	15,290	49,810	11,772	794	12.57
50,000	302	16,556	51,047	11,754	765	12.52
60,000	272	18,382	54,877	12,140	700	12.84
70,000	242	20,664	59,092	12,687	690	13.38
80,000	206	24,271	64,994	13,439	686	14.13
100,000	142	35,211	82,154	15,396	622	16.02
150,000	65	76,923	136,592	19,808	540	20.35

4.3.1 Performance of distance computation

Table 1 shows the processing times for a calculation chunk using the presented CPU and GPU approaches. The evaluation is performed for calculation chunks that differ in the number of target and reference points. Consequently, the total number of distance calculations increases with a growing amount of data. The CPU implementation shows the worst performance. In addition, using multiple CPU cores is still significantly slower than the GPU approaches. The CUDA implementation performs best independently of the size of the calculation chunks and is used for further performance evaluation.

The second evaluation scenario is related to the processing of an in-memory chunk using the CUDA implementation and is shown in Table 2. The used in-memory chunk contains five million target and reference points and is organized in the multi-data octee with a leaf capacity of 256 points for each data set. A larger capacity increases the number of distance computations for calculation chunks, because more reference points are selected from intersected leaf nodes for the computation. A lower capacity would result in more effort to evaluate and load

Table 3 Performance of the overall change detection workflow in minutes

Task	Time	Percent
Create multi-data octree	1,260	61.1%
Load In-memory chunks	114	5.5%
Determine calculation chunks	63	3.1%
Distance computation on GPU	624	30.3%
Overall change detection	**2,061**	**100%**

the tree structure. The multi-data octree is used to select the calculation chunks for the distance calculation and allows control of the size and total number of target and reference points in the calculation chunks. Smaller calculation chunks make the selection and preparation of the calculation chunks computationally more expensive, but also reduce the total number of distance computations for all calculation chunks. The optimal calculation chunk size depends on the processing scheme and the available hardware capabilities. For our system the CUDA implementation performs best using calculation chunks with a maximum size of approximately 50,000 target points.

4.3.2 Performance of workflow

The performance is evaluated with a real-world data set as a case study. Two 3D point clouds of the urban area of the city of Frankfurt am Main are used to evaluate the performance of the overall change detection workflow. The target cloud is an airborne scan of the year 2009 with 7.1 billion points. The reference cloud was captured in the year 2005 and contains 1.9 billion 3D points. Both scans cover almost the same urban area but differ in their sampling density.

The overall time to attribute the target point cloud can be divided into sub-processes to evaluate the different time demands as illustrated in Table 3. The process to create the multi-data octree takes the most time, due to the frequent access of secondary storage to organize the data and prepare the multi-data octree. The time to load the in-memory chunks and to determine the calculation chunks is quite negligible, because of the fast access to the data provided by the multi-data octree. The distance computation for all calculation chunks takes about 10.5 hours on the GPU and is significantly faster than a CPU implementation that would require several months for the computation. The overall change detection takes about 35 hours and is a reasonable preprocessing time for two data sets with a total of nine billion points and a size of 135 GB. The sub-process to create the multi-data octree has optimization potential using multiple storage systems and parallel processing. This, however, is beyond the scope of this work. Moreover, the performance of the distance computation can be increased by distributing the computation over multiple GPUs.

5 Visualization of the Results

In this section, we present our interactive out-of-core rendering system for attributed 3D point clouds. As usage scenario we use the attributed 3D point cloud of Frankfurt am Main presented in Section 4.3.2.

5.1 Rendering System

The rendering system has to cope with 3D point clouds of arbitrary size. In addition, it is necessary to visualize the 3D point cloud interactively with the ability to customize the stylization of the visualization, e.g. depending on the change detection results during the rendering process. To fulfill these requirements we extended the out-of-core real-time point based rendering system of Richter and Döllner (2010). In contrast to the multi-data octree presented in Section 4.1, a multi-resolution octree is used that is designed to handle single 3D point clouds. Instead, the multi-resolution octree manages several levels of detail (LoDs) of a 3D point cloud to enable an adaptive visualization.

The multi-resolution data structure is based on an octree and uses a bounding sphere hierarchy to perform culling and selection of relevant nodes for the rendering. The 3D points and their attributes are stored in the leaf nodes. To store a LoD for a certain spatial volume, inner nodes of the data structure are utilized storing average position, spatial volume, color, and distance attributes. The traversal of the data structure is controlled by a threshold parameter determining an adequate LoD for the rendering process by projecting each node's bounding sphere into screen space. If the projection is below the threshold (e.g. 5–10 pixels), no further traversal of the tree part is performed and the node is selected for rendering. Finally, all selected inner nodes are rendered using the bounding sphere's center and all selected leaf nodes are rendered using the assigned 3D points. Due to the size of the data structure, only parts that are visible and relevant to the visualization are kept in main memory. The out-of-core functionality is implemented by a worker thread that loads required parts of the data structure into main memory if higher resolution and more LoDs of the 3D point cloud are required during user interaction.

5.2 Visualization of Attributed 3D Points

A challenge for rendering massive 3D point clouds at interactive frame rates are the limited storage capacities of main memory and GPU device memory. The size of the rendered data affects the performance of the rendering process. Each point of the 3D point cloud is given by a three-dimensional floating point vector with a size of 12 bytes and optional attributes, such as color, distance, and normal data. To keep the data compact and ensure properly aligned memory, we limit the additional data per point to 4 bytes, resulting in 16 bytes per 3D point. Possible choices for the additional data are color values from aerial images, distance values with float precision resulting from the change detection, or a combination of color and distance values with reduced precision. Colors from aerial images together with distance information lead to an effective visualization tool for change detection results.

The results of the distance computations are mapped to color attributes. This mapping is configurable to allow for the adaption of the visualization depending on the application and analysis task. Figure 5 shows the processing and rendering results of the entire data set from different points of view. The entire data set with color information from aerial images without distance information is shown in Figure 5a. In Figure 5b–d all points that most probably changed are highlighted with a color gradient from green to red depending on the degree of change. 3D points with a distance value below one meter are classified as points that have not changed and are rendered with colors from aerial images to enable a better perception of the 3D point cloud structure. The detailed view in Figure 5d shows buildings and sites that are

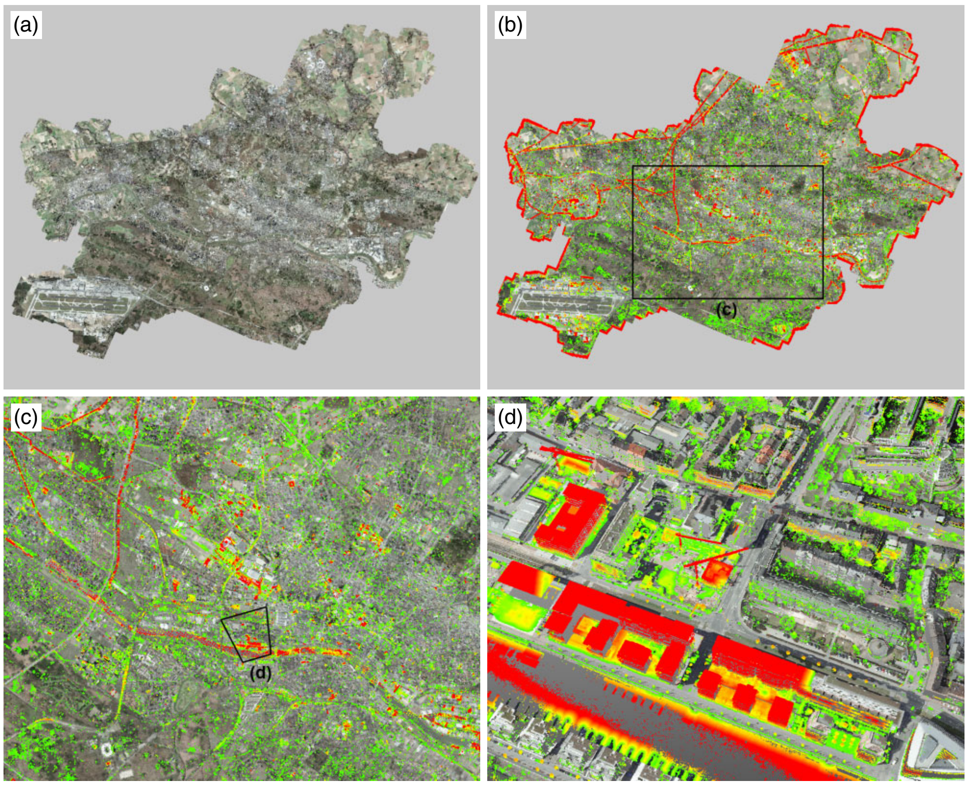


Figure 5 Change detection results of a 3D point cloud with 7.1 billion points of the urban area of a city: (a) Visualized with color information; (b) Visualized with color and distance information. Points with a distance below 1 m are rendered with color information from aerial images and points with distance between 1 and 5 m are rendered with a gradient from green to red; and (c)–(d) Two closeup views of (b)

clearly detected as changes. Different configurations of the visualization are illustrated in Figure 6. The adaptive coloring of 3D points based on their distance values, allows the highlighting of structures for different application tasks, e.g. finding changed buildings and sites (Figure 6c) or vegetation (Figure 6d).

By swapping the target and reference clouds, additional analysis is possible as illustrated in Figure 7, where a construction site was captured. The left column shows a target cloud from 2009, attributed with a reference cloud from 2005. The right column shows the results of the change detection using the data set from 2005 as the target cloud and the data set from 2009 as the reference cloud. In the upper row the 3D point clouds are attributed with color attributes from aerial images, whereas in the middle row a black-to-white color gradient is used to visualize 3D points based on their height. This is helpful for recognizing even small structures, e.g. cars and vegetation that are more difficult to recognize in the 3D point cloud colored by aerial images. In the lower row the change detection results are shown. In Figure 7e new structures are shown and in Figure 7f structures that disappeared are recognizable.

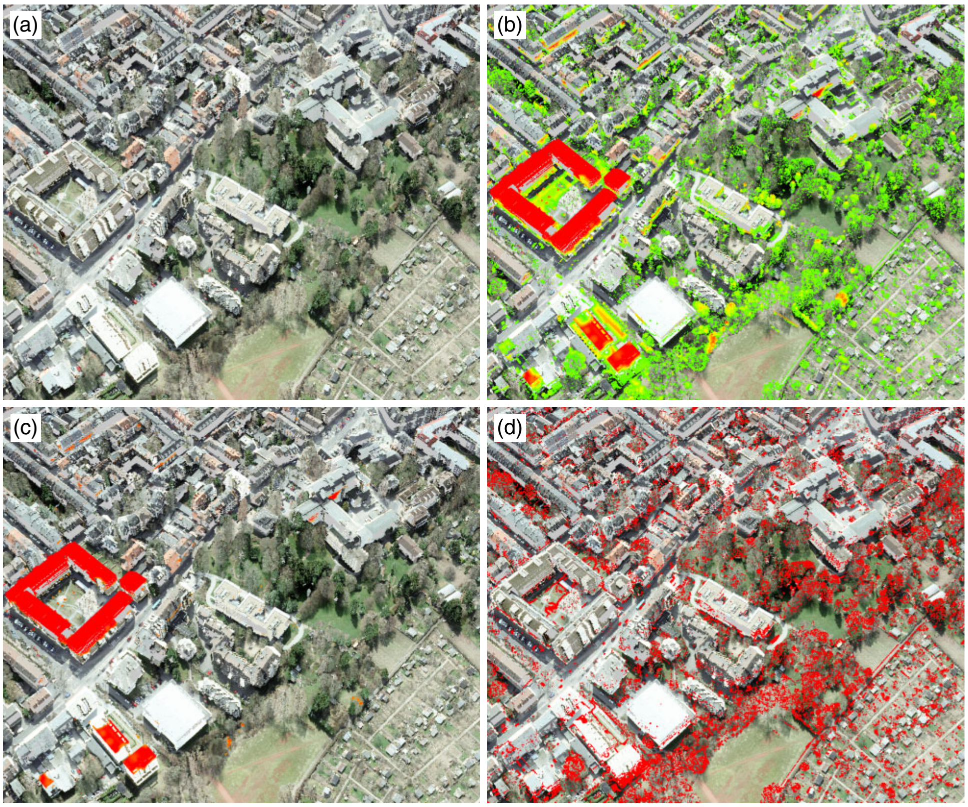


Figure 6 Change detection results visualized with different settings: (a) Original 3D point cloud; (b) All changes above 1 m highlighted; (c) Major changes above 7 m highlighted, e.g. for buildings; and (d) Minor changes between 1 and 2 m highlighted, e.g. for vegetation

6 Conclusions and Future Work

In this article, we have presented and discussed a concept and implementation for detecting changes in massive 3D point clouds without having to reduce or rasterize the raw data sets. To partition the data into smaller parts suitable for parallel processing, we introduced a specialized out-of-core data structure that enables fast and efficient access to multiple 3D point clouds. We discussed several parallel processing schemes, with the GPU-based approaches turning out to be clearly superior. In particular, the CUDA-based implementation showed the overall best results. Our approach enables new applications that take advantage of the full resolution and potential of 3D point clouds. We also introduced techniques and concepts for the real-time visualization of the results, enabling the exploration and analysis of temporal changes of highly accurate 3D points clouds of arbitrary size. As part of our future work, we will investigate how to incorporate semantic information into the change detection process. This would enable more precise analysis of points assigned to different sub-classes, such as vegetation, terrain, buildings, and sites.

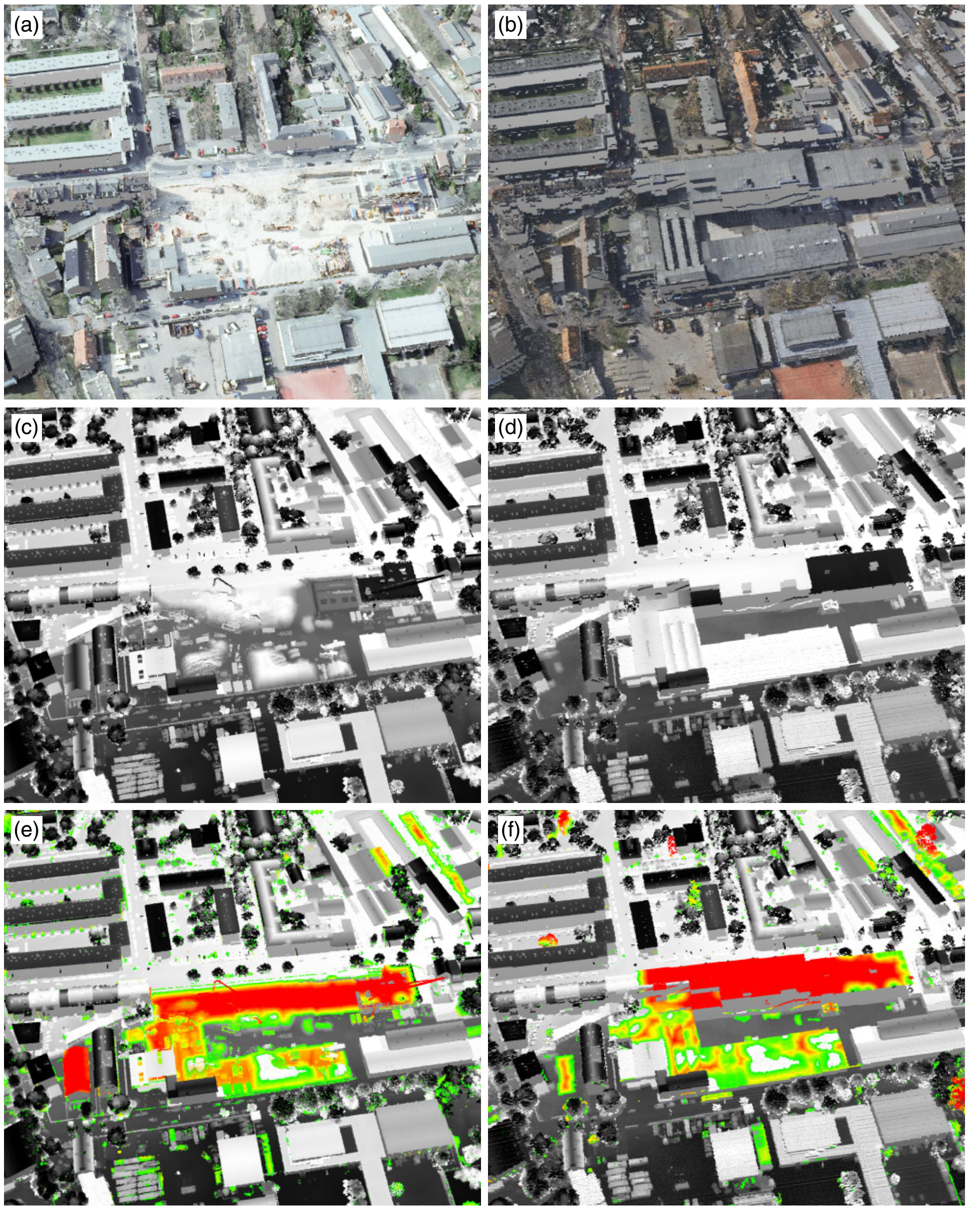


Figure 7 3D point clouds and change detection results for scans from 2005 and 2009. The first row shows the 3D point cloud with color information from aerial images, the second row uses a color gradient based on the 3D point height and the third row shows the degree of change with a color gradient from green to red based on the distance values above 1 m

References

Barber D M, Holland D, and Mills J P 2008 Change detection for topographic mapping using three- dimensional data structures. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 37(B4): 1177–82

Besl P J and McKay N D 1992 A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14: 239–56

Botsch M, Hornung A, Zwicker M, and Kobbelt L 2005 High-quality surface splatting on today's GPUs. In *Proceedings of the Second IEEE/Eurographics Symposium on Point-Based Graphics*, Stony Brook, New York: 17–24

Butkiewicz T, Chang R, Wartell Z, and Ribarsky W 2008 Visual analysis and semantic exploration of urban LiDAR change detection. *Computer Graphics Forum* 27: 903–10

Farber R 2011 *CUDA Application Design and Development*. San Francisco, CA, Morgan Kaufmann Publishers

Girardeau Montaut D, Roux M, Marc R, and Thibault G 2005 Change detection on points cloud data acquired with a ground laser scanner. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Science* 36(3): 30–5

Gröger G and Plümer L 2009 Updating 3D city models: how to preserve geometrictopological consistency. In *Proceedings of the Seventeenth ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Seattle, Washington: 532–35

Heinzel S, Guennebaud G, Botsch M, and Gross M 2008 A hardware processing unit for point sets. In *Proceedings of Graphics Hardware '08*, Sarajevo, Bosnia, Herzegovina: 21–31

Kang Z and Lu Z 2011 The change detection of building models using epochs of terrestrial point clouds. In *Proceedings of the International Workshop on Multi-Platform/Multi-Sensor Remote Sensing and Mapping (M^2RSM)*, Xiamen, China: 1–6

Kreylos O, Bawden G W, and Kellogg L H 2008 Immersive visualization and analysis of LiDAR data. In *Proceedings of the Fourth International Symposium on Advances in Visual Computing*, Las Vegas, Nevada: 846–55

Lafarge F, Descombes X, Zerubia J, and Pierrot-Deseilligny M 2010 Structural approach for building reconstruction from a single DSM. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32: 135–47

Leite P, Teixeira J, de Farias T, Teichrieb V, and Kelner J 2009 Massively parallel nearest neighbor queries for dynamic point clouds on the GPU. In *Proceedings of the Twenty-first International Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil: 19–25

Levoy M and Rusinkiewicz S 2000 Qsplat: A multi-resolution point rendering system for large meshes. In *Proceedings of the Twenty-seventh International Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH)*, New Orleans, Louisiana: 343–52

Luebke D, Harris M, Krüger J, Purcell T, Govindaraju N, Buck I, Woolley C, and Lefohn A 2005 GPGPU: General purpose computation on graphics hardware. In *Course Notes of the Thirty-second International Conference on Computer Graphics and Interactive Technologies (SIGGRAPH 2005)*, Los Angeles, California

Matikainen L, Hyyppä J, and Kaartinen H 2004 Automatic detection of changes from laser scanner and aerial image data for updating building maps. *International Archives of Photogrammetry, Remote sensing and Spatial Information Sciences* 35(B2): 434–39

Mémoli F and Sapiro G 2004 Comparing point clouds. In *Proceedings of the Second Eurographics Symposium on Geometry Processing*, Nice, France: 32–40

Monserrat O and Crosetto M 2008 Deformation measurement using terrestrial laser scanning data and least squares 3D surface matching. *ISPRS Journal of Photogrammetry and Remote Sensing* 63: 142–54

NVIDIA Corporation 2011 *NVIDIA CUDA C Programming Guide v4.0*. Santa Clara, CA, NVIDIA Corporation

Pfister H, Zwicker M, Baar J V, and Gross M 2000 Surfels: Surface elements as rendering primitives. In *Proceedings of the Twenty-seventh International Conference on Computer Graphics and Interactive Technologies (SIGGRAPH 2000)*, New Orleans, Louisiana: 335–42

Richter R and Döllner J 2010 Out-of-core real-time visualization of massive 3D point clouds. In *Proceedings of the Seventh International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (Afrigraph '10)*, Franschhoek, South Africa: 121–28

Sanders J and Kandrot E 2010 *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, MA, Addison Wesley

Schneider D 2006 Terrestrial laser scanning for area-based deformation analysis of towers and water damns. In *Proceedings of the Third IAG Symposium of Geodesy for Geotechnical and Structural Engineering and Twelfth FIG Symposium on Deformation Measurements*, Baden, Austria

Snavely N, Garg R, Seitz S M, and Szeliski R 2008 Finding paths through the world's photos. *ACM Transactions on Graphics* 27: 11–21

Snavely N, Seitz S, and Szeliski R 2006 Photo tourism: Exploring photo collections in 3D. *ACM Transactions on Graphics* 25: 835–46

Trinder J and Salah M 2011 Airborne LiDAR as a tool for disaster monitoring and management. In *Proceedings of the GeoInformation for Disaster Management Conference (Gi4DM 2011)*, Antalya, Turkey

Van Gosliga R, Lindenbergh R, and Pfeifer N 2006 Deformation analysis of a bored tunnel by means of terrestrial laser scanning. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 36(5): 167–72

Vosselman G, Gorte B, Sithole G, and Rabbani T 2004 Recognising structure in laser scanner point clouds. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 46(8): 33–8

Wimmer M and Scheiblauer C 2006 Instant points: Fast rendering of unprocessed point clouds. In *Proceedings of the IEEE/Eurographics Symposium on Point-Based Graphics*, Prague, Czech Republic: 129–36

Zhou Q Y and Neumann U 2008 Fast and extensible building modeling from airborne LiDAR data. In *Proceedings of the Sixteenth ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Irvine, California: 1–8